Abstract—This paper proposes an approach to simulate the
function of the control part of a Grafcet model, translating it into
C code in a Unix environment. First, the Grafcet/C generation
schemes are established. The Grafcet model, described in graphic
or text form, is transformed into an internal form and then to C
code by a generation algorithm based on the previously found
diagrams. The result is a program that simulates the operation
of the automation in question and makes it possible to validate the
functional specifications of sequential automation. This validation
can be used for educational purposes, such as the learning of the
Grafcet formalism, or corrective or evolutionary maintenance.
Once the configuration, testing, and validation of the program
are complete, it is possible to implement the object code on the
microcontroller of the control system.

Keywords—C/Unix; Grafcet; process; sequential automation

I. INTRODUCTION

Equipment manufacturers and automation engineers
responsible for automated industrial installations should master
the programs that drive their installations to perform preventive
[1, 2], corrective, adaptive, and evolutionary maintenance tasks
[3]. These programs are often outsourced, use a variety of
libraries, and depend not only on the problem to be solved but
also on the past of their provider. Grafcet [4, 5] is a graphic
formalism for describing automatisms, accepted by mechanical
automation engineers who consider it to represent the right
level of specification without much complexity. In an
automated system, the control part is the image of the operative
part that represents the automated machine. Simulation and
validation of the operation of an automated system are
necessary before its implementation in the actual installation.
Simulating the operation of Grafcet is equivalent to translating
it into appropriate languages, which generate programs with the
same semantics. These programs have shown their usefulness
for maintenance or educational purposes. This study chose
C/Unix as the target language and system for the Grafcet
translation. This choice was motivated by two reasons; the C
language extended by the Unix libraries has all the necessary
tools for the translation of Grafcet, and the required hardware
and software configuration for the application is very simple,
just a personal computer with a Linux distribution.

II. RELATED WORK

Several studies achieved to formalize Grafcet. In [6],
Grafcet translation schemes were designed in the Occam2
language, which is executable on transputers, exploiting the
possibilities of expressions of parallel tasks offered in Occam2
to express the respective representation in Grafcet. The
resulting program could run on a parallel machine and simulate
the actual operation of an automated system. This work could
obtain the equivalent Occam2 program from a Grafcet in
graphical or textual form. Being a parallel language, Occam2
possesses the necessary tools to translate Grafcet, but the
resulting program can only be exploited if a parallel machine
existed. Since parallel machines are only available in certain
research laboratories or specific industrial settings, such a
program may have limited use. In [7], a semi-coarse ontology
was defined and tested by integrating it into an existing
educational tool to teach Grafcet for use in programmable logic
controllers. This ontology was OWL (Web Ontology
Language) DL based, a specific decidable fragment of first
order logic applied to OWL markup language. The objective of
this method was to complement previous studies and promote
this type of technique in the formalization of Grafcet. The
advantages of ontologies are numerous, as they make
collaboration and sharing of knowledge easier, provide better
reliability, and assure to handle automatically any input change
without having to recompile the processing code. This
approach was validated according to two criteria, accuracy and
completeness. A new vision was adopted in [8], by proposing a
systematic implementation of the control software in IEC
61499. This constituted a key advantage over previous Grafcet
implementations because it allowed engineers to implement
models distributed over several devices and also kept the initial
centralized design. IEC 61499 has all the translation tools for
most of the Grafcet elements. This work made it possible to
systematically translate Grafcet and introduced several
translation models. The disadvantages of this method lied in the
fact that it was not possible to model the structuring
mechanisms such as fences or forcing steps, and the macro-
steps that could be implemented were limited to simple
sequences.

Received: 19 September 2022 \ Revised: 3 October 2022 \ Accepted: 6 October 2022
### III. MOTIVATION FOR CHOOSING C LANGUAGE

Grafcet is a logical automatism description formalism that allows expressing competing processes. A language to translate Grafcet has to preferably be parallel [6] and/or real-time. The duration of the task's installation or its switching time is decisive in choosing such a language. C and Unix [9] were chosen due to their portability, universality, and control by most computer scientists. Its disadvantage lies in the fact that Unix is not quite real-time, because its slow temporal primitives were defined according to the only problem of the timeshare. C language is a High-Level Machine-Oriented Language (HLMOL) that allows defining bit-close fields, and expanded by Unix libraries can provide the illusion of simultaneous execution of tasks on uniprocessor machines, as shown by functions such as fork, wait, sleep, kill [9]. Pipes are the main means of communication between Unix processes [9]. Several synchronization means are available in Unix. This study used the wait function, which is the most basic mean of synchronization and can be used to synchronize a parent process on the termination of its children. Time management was carried out using the sleep() function, as the call to the sleep(n) function suspends the calling process for n seconds. Since the seconds are not useful for many real-time scenarios, the macro tempo() was used to allow timers in microseconds:

```c
#define tempo(n){
    clock_t  reveil = clock()+n;
    while(clock()<reveil) sleep(0);
    } /*n in microseconds */
```

So, sleep(1) is equivalent to tempo(1000000). These timers were used in processes running in parallel and can limit the waiting time for certain events.

### IV. SIMULATION SYSTEM DESCRIPTION

Two subsystems constitute the simulation system [10]:

- **Grafcet entry:** this subsystem offers two possibilities:
  - Graphical input: Based on a graphical editor, allows any Grafcet to be entered graphically and outputs its image data structure.
  - Text entry: Rarely used in practice, except for maintenance purposes. It allows to enter the Grafcet as text, so requires a text analyzer that outputs the same data structure as the graphical editor.

- **Translation of the Grafcet:** This subsystem exploits the data structure from the entry and translates it into C. Two steps are possible:
  - Interpreter/simulator: A program that executes step-by-step the appropriate C sequences according to the data structure.
  - Generator: A program that creates a complete C code of all the Grafcet. Executing this code is the simulation of the automatism described by the Grafcet, as long as this code is not configured according to real I/O.

In both cases, translation requires defining the Grafcet/C generation schemes.

### V. GRAFCET/C TRANSLATION SCHEMES

The definition of the translation schemes from Grafcet to C consists of finding for each Grafcet basic element a program scheme in C which has the same semantics. The elements of Grafcet are: simple transition, divergences (AND, OR), convergences (AND, OR), the stage, and the macro stage [10].

#### A. Preliminary Study [10]

Let's consider the following scenario:

- A rotating bar at a position x, y of the screen successive display in x, y of characters -, /, \, |, \, -, /, …

- In case of no overflows, pressing the arrow keys causes the bar to move down, left, right, and up respectively.

- If limits are exceeded, the above characters produce an audible signal, and the bar keeps rotating in the same place.

- Pressing the character "q" stops the scenario.

- Pressing any other character is ineffective.

Figure 1 shows the Grafcet of the above scenario.

![Grafcet associated with the rotating bar scenario.](image)

Let EB and TB be the stages and transitions of the rotating bar branch, and EL and TL be the stages and transitions of the reading characters branch.

- E0: initial stage, beginning of the program.
- EB1: rotating bar.
- EB2: end of the rotating bar process.
- EB3: Test of the character read in the pipe.
- EB4: audible alarm.
- EB5: moving the bar.
- EB6: rest (sleep).

**Stages**

- EL1: read and test the read character.
- EL2: sending a KILL and exit.
- EL3: write \, /, \, /, \, /, … in the pipe.
- EL4: rest (sleep).
- TB1: reception of a KILL signal.
- TB2: read \, \, \, →, →, \, in the pipe.
- TB4: overcoming limits.

**Transitions**

- TB5: no overflows.
- TB6: = 1; TB7 = 1
- TL2: read \, \, \, \, \, or → from the keyboard.
- TL3: reading of the \, \, character of the keyboard.
- TL4 = 1; TL5 = 1; TLF = 1.
B. Principles of the Image Program

Once the program is launched, two processes are created and launched simultaneously (proc1, proc2). Proc1 is a standalone process associated with the rotating bar, while proc2 is associated with its movement and is under external influence. At some point, the two processes must act simultaneously on the coordinates \( x \), \( y \) of the bar. A first solution is to create a critical section within each process to achieve mutual exclusion of both accesses [8]. A second solution would be to allow effective access to a single process, such as proc1, which will rotate and move the bar, proc2 reads the arrow characters, considered as the move commands of the bar, and sends them through a pipe to proc1 to use them to move the bar in the desired direction. The end of the scenario will take place when proc2 reads the ”q” key. Table I shows the general processing algorithms of proc1 and proc2. This program, modeled in Grafcet, is the basis to deduce the basic translation schemes from Grafcet into C. These schemes were extended for industrial processes where the system entries can be numerous, simultaneous, and real-time. Therefore, push buttons and sensors were integrated.

<table>
<thead>
<tr>
<th>TABLE I. PARALLEL RUNNING OF PROC1 AND PROC2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROC1</strong></td>
</tr>
<tr>
<td>At each position of the bar:</td>
</tr>
<tr>
<td>Display the bar</td>
</tr>
<tr>
<td>Look in p[0] if a character is placed in the pipe.</td>
</tr>
<tr>
<td>If yes the character is stored in cc1 and tested.</td>
</tr>
<tr>
<td>If exceeding the limits, warning sound, otherwise move in the desired direction.</td>
</tr>
<tr>
<td>Rotation of the bar.</td>
</tr>
</tbody>
</table>

C. Grafcet-C Translation Schemes

A simple transition is an external event that can arise at any time from a sensor, button, etc. Three cases exist to simulate it: keyboard, order box, and always concurrent. Keyboard input can be given as a simple scheme:

\[
\text{cc}=\text{getch()} \]

In case of an order box, let boitcom be the port address of the command box, reading 8 to 16 digital inputs. By assimilating all the bits to zero in the absence of an entry, the scheme could be the following:

\[
\text{while}(!(*\text{boitcom})) \text{sleep}(0); \\
\]

These diagrams are taken up and developed when a configuration language would cover the essential cases concerning the real-time management of arbitrary devices, which will be part of a realistic simulation. Meanwhile, the input from the control box can be simulated by programming the keys of the keyboard, associating a sequence of bits: for \( i=1,...,7 \), command[0]... command[7], command[i]=1 when the receptivity \( i \) is true, and command[j]=0 when no receptivity is true. In an always occurrence case, \( t=1 \), which means that the sum of the internal and external receptivity conditions is 1. In this case, a comment is generated \( (/^* t=1 */^) \).

1) Divergences

- **AND divergence:**
  
  \[
  \text{while}(t) \text{sleep}(0); \\text{rep}=\text{forkn}(/\text{tab_fonc}.nproc.\text{tab_pid}); \\text{repf}=\text{wait}(/\text{status}); \\text{while}(\text{rep1} !=1) \text{wait}(/\text{status}); \\
  \]

  where \( \text{tab_fonc} \) is an array of pointers to functions performed in the context of each process, \( nproc \) is the number of the created processes, \( \text{tab_pid} \) is the pid chard of the created processes, and \( \text{tab_fonc[i]} \) is the pointer to the function executed by the process with the pid \( \text{tab_pid[i]} \). \( \text{Forkn}() \) details can be found in [9].

- **OR divergence:** The exclusive OR divergence is given by:

  \[
  \text{while}(\text{t1}&\&\text{t2}&\&\&\&\text{t3}... &\&!\text{tn}) \text{sleep}(0); \\text{switch(\text{transition})}{ \\text{case} \ t1 : \text{p1}(); \text{case} \ t2 : \text{p2}(); \... \text{case} \ tn : \text{pn}(); } \\
  \]

  where \( \text{pi}() \) is the procedure executed for the transition \( ti \) and there is no priming of new paths. Inclusive OR is given by:

  \[
  \text{while}(\text{t1}&&\text{t2}&&...&&!\text{tn}) \text{sleep}(0); \\text{if}(\text{t1}) \text{p1}(); \\text{if}(\text{t2}) \text{p2}(); \... \\
  \text{if}(\text{tn}) \text{pn}(); \text{*/ the creative process enters the zombie state */} \\text{rep1=wait(&status)}; \text{repf=wait(&status)}; \text{while}(\text{rep1} !=1) \text{wait(&status)}; \\
  \]

  where \( \text{pi}() \) is the process creation procedure \( i \). There is a boot of new paths. Two images are likely in the classical case of an input form: A general but expensive image (competition diagram), or an effective image applicable under certain conditions. In the case of an effective image, the alternative scheme is applicable if and only if the transitions are disjoined two by two.

The evaluation of this question in a generator is the subject of a decision procedure, which in case of difficulties may substitute, the absolute criterion above, one or more sufficient conditions easier to evaluate. For example, if two transitions occur as products of elementary conditions, they are disjoint if the same condition is present in the two transitions in opposite forms, such that: \( T1 = x y z \) and \( T2 = x y z \). In the case of expectations with time-out, there is divergence with two issues, one of which carries the receptivity “event” and the other a receptivity “time limit”. Since this can only be performed in excess, it is normal to consider the two as disjoint (in case of conjunction, the event is considered to have happened after the prescribed duration). The quality of the decision procedure thus directly governs the quality of the code, which may be inaccurate, correct but heavy, correct and effective, and even optimal for a perfect decision procedure. On the other hand, the generation with divergences must agree with the generation with convergences.

2) Convergences

- **AND Convergence:** Two cases arise: If pi has the same divergence as the origin, the synchronization is performed by `wait()`, while if they don’t have the same origin, synchronization is mandatory. For each father process, `synchro` is a global variable. Initially, `synchro` is the number of incoming branches. Once it finishes, each process simulating an incoming branch must access `synchro` and decrement it. When `synchro` becomes null, it implies the end of all branches. As the access to `synchro` may be simultaneous, it must therefore be a critical section within each process, using either semaphores or locks [11]. This study used a method of choosing a process, called the coordinator, and took the one that simulates the most left incoming branch. When it finishes, the coordinator performs the following algorithm.

```c
/* scheme associated with coordinator*/
/*only the coordinator accesses synchro*/
int rep; char cc;

/* as soon as it finishes, it decrements synchro */
synchro--; 
while(1)
{
    cc = ' '; 
    dup(tube1[1]);
    close(tube1[1]);
    read(tube1[0],cc,1);
    if (cc=='f') synchro--;
    ...
    cc=' ';
    dup(tubef[1]);
    close(tuben[1]);
    read(tuben[0],cc,1);
    if (cc=='f') synchro--;
}
/* synchro=0, the process p0 kills all the other processes and makes an exit */
rep=kill(pid1, SIGKILL);
rep=kill(pid2, SIGKILL);
... 
rep=kill(pidn, SIGKILL);
exit();
```

Once it finishes, each other process associated with other branches should perform the following algorithm:

```c
/* let's suppose the process number i, other than the coordinator, sends the character 'f' in the pipe */
dup(tubei[0]);
close(tubei[0]);
write(tubei[1],'f',1);
/* infinite loop as wait */
for( ; ;) sleep(0);
```

- **OR Convergence:** Exclusive OR comes down to the simple transition, while inclusive OR is discussed in the same way as the AND convergence.

3) The Stages

It executes within a process, and can be simulated by a message specifying it, possibly a time-dependent timer associated with it, encapsulated in an `enn()` procedure:

```c
void enn()

printf("etape %d",num_etape);
tempo(duree) ;
```

4) The Macro Stage

The macro stage is translated using a procedure that is an image similar to the main program because it is a sub-Grafcet and militates a recursive generation.

5) Forcing

The diagrams can be implemented in the case where the automatism is modeled by a single Grafcet, which is a single connected component. They can be extended by adding the associated macros in the case of forcing or applied in the case of a hierarchy of Grafcets. The forcing function is an action of macro stage M. This is then called macroaction, and is a procedure using another Grafcet (slave). The functions associated with forcing operations are summarized below:

- **Freezing:** This operation consists of sending the signal SIGSTOP to the active stages of the given Grafcet.

```c
void suspend(g)
int i, rep;
i=1 ;
while (i<= nbactif){
    rep=kill((pid[i],SIGSTOP);
    i++ ;
}
```

- **Disabling the slave Grafcet:** This operation is associated with a `deactivate()` procedure which consists of sending a SIGKILL signal to the active processes in the slave Grafcet given by g.

- **Put in initial situation:** This procedure consists of calling a subroutine that is analogous to the main program. This is similar to using the macro stage in the case of the normal operation of automation.

- **Put in any situation:** Two cases may arise, reactivate a previously suspended Grafcet, which would consist of sending the SIGCONT signal to the suspended process, or activate certain stages of the Grafcet in question where it would be necessary to put to true their input receptivities and launch the functions associated with them.

VI. NECESSARY CONFIGURATION FOR THE SYSTEM

The implementation of this system requires:

- A Unix or Linux or any multitasking system.
- A graphic screen for entering the Grafcet.
- A keyboard as input device and, if possible, a box of commands.
- Effectors: LEDs, bulbs, effectors.

The screen is divided into two windows. The first window (FEN1) is used to draw the currently active Grafcet (command part), while the second (FEN2) is specific to the messages that illustrate the actions carried out by the operative part. Initially,
the Grafcet is animated from the initial situation. The arrival of the transitions from the keyboard or the control box causes the evolution of the Grafcet, i.e. activation of the new stage or stages (following the transition), and deactivation of the stage or stages. Active stages will be highlighted in FEN1, showing their associated messages in FEN2. The command box allows making several entries at once, a case that can be tested for OR divergences and transitions whose branches run in parallel. Figure 2 shows a Grafcet and the corresponding C program.

Figure 2.
From Grafcet to C code.

VII. APPLICATION

Figures 3 and 4 show two industrial Grafcet examples that were used to validate the proposed method. The principle of validation of a Grafcet has two phases: static validation, respecting the conditions of good form, and dynamic validation of the execution.

A. Grafcet Good Form Conditions

A Grafcet is:

- Except: if for any situation accessible from E0, no step is reactivated.
- Living: If for any accessible situation from E0, a crossing sequence of any transition exists.
- Clean: If for any situation accessible for E0, there is a crossing sequence leading to E0.

B. Importance of the Preceding Properties for the Grafcet Evolution

If the Grafcet is safe, no step is reactivated during its possible evolutions. Reactivation is dangerous and can lead to errors. A living Grafcet will never block and will never find inert steps or transitions (unactivated steps, unsensitized transitions) at a certain stage of the evolution. If a Grafcet is clean, this implies possible re-initiation. This is a very important phenomenon for automation, as the initial step is considered a resting stage. The proposed method assumes:

- Transitions from the keyboard or the control box are always occurring (= 1), simulating well external or internal events.
- Simple steps: The considered Grafcet can easily be assimilated to the complete Grafcet that takes into account any type of transition or step. This method is effective for Grafcet validation.

The Grafcet shown in Figure 3 is not safe because there is a reactivation of step 2, in the case of crossing transition 5. The proposed method indicates it by a message and rejects it when meeting again step 2 within the normal operating cycle.

In Figure 4, transitions 2 and 3, are OR divergences. If they occur together, the OR of this divergence is inclusive and the Grafcet is clean, alive, and safe. If they do not occur together, the OR is exclusive, and one branch will be executed, but arriving at the AND convergence, transition 5 can be crossed only if steps 4 and 5 are both active and the transition is equal.
to 1, which is not the case if the OR is exclusive. The proposed program reports a blockage in this case.

C. Compatibility of Simultaneous Actions and Correlation of Stages and Receptivity

A subsystem displaying the steps and associated actions would allow the user to know if the steps that have run simultaneously correspond to compatible actions, after observing the execution of the Grafcet. For example, two pumps, one operating while the other is at rest, should never appear together in two simultaneous actions in a Grafcet. Similarly, the action-receptivity correlation can be checked by consulting the transitions and the associated receptivity conditions (the symbol table contains all the information relating to the steps and transitions).

D. Search of Cycles

In principle, a cycle exists, outside the normal operating cycle, if there is a reactivation of at least one previous step. The proposed algorithm, as designed, rejects the Grafcet as soon as a previously activated stage is reactivated. A cycle within the proposed algorithm, as designed, rejects the Grafcet as soon as a cycle, if there is a reactivation of at least one previous step. The work would focus on translating Grafcet into Promela/SPIN.

REFERENCES